|  |  |
| --- | --- |
| 991CN X  Make 991CNX Great Again | fx-es(ms)编写组 |

# 前 言

目 录

前 言 1

第一章 汇编语言 4

1.1 汇编语言简介 4

1.2 常用指令（nX-U16指令集） 5

第二章 硬件与系统架构 12

2.1 nX-U16处理器 12

2.2 内存与寄存器 12

2.3 自检与计算器版本号 12

2.4 ROM概述 12

2.5 矩阵键盘 12

第三章 字符与表达式 15

3.1 单字节与双字节字符 15

3.2 数学显示字符 16

3.3 映射字符 17

第四章 内存分区 17

4.1 输入区，回放区与撤销区 17

4.2 变量区与变量存储格式 19

4.3 历史记录区 19

4.4 设置区 19

4.5 即时状态区 19

4.6 堆栈区 19

4.7 随机数种子 19

4.8 显示缓存区 19

4.9 空闲区 19

4.10 杂项 19

第五章 输入的显示与编辑 19

5.1 输入的显示 19

5.2 输入的编辑 19

第六章 特殊字符 19

6.1 字符转换器 19

6.2 溢出 20

6.3 @法刷字符 20

6.4 字符的特殊性质 20

第七章 返回导向编程 20

7.1 an/@的异常性质 20

7.2 ROP进入方法 20

7.3 简单ROP 20

7.4 ROP进阶 21

7.5 ROP实战 21

第八章 ROM刷机 21

附录一 字符表 21

附录二 常用汇编指令 22

附录三 按键，键盘码，内码表 22

附录四 模拟器的配置与使用 22

附录五 ROM函数地址索引 23

附录六 用Ghidra反编译ROM 23

附录七 Github上的数字资源仓库 23

结 语 23

# 汇编语言

* 1. **汇编语言简介**

这一章的内容可能较为复杂，但它是之后编程的基础。对没有系统熟悉过汇编语言的同学，这一章和下一章或许较为枯燥，不妨同时读读第三章。它的内容与本章基本无关，但相对有趣一些，也更有可操作性。

* + 1. **CPU的工作原理**

假设有个小学生，正在照着数学书，在草稿纸上练习竖式乘法。从某种程度上来说，他就像是一个正在工作的**CPU**（Central Processing Unit，中央处理器）。**ROM**（Read-Only Memory，只读存储器）就是他面前的数学书（上面写着他正在运行的程序）以及他铅笔盒里贴着的乘法表（也就是程序运行所需要的数据）。两者的共同特点是，它们都只能读取，不能修改；不论程序的输入是什么，它们都是固定不变的。**RAM**（Random Access Memory，随机存储器，通称内存）就是他手上的草稿纸，纸上写着程序产生的一些数据。它的特点是，可以读也可以写，而且能够存储大量的数据。但是，CPU（这个小学生）不能直接操作RAM（纸）上的数据，而必须通过眼睛读数，用手写数。CPU可以根据ROM上的指令做一些操作，也可以从ROM或RAM中读取数据，还能向RAM里写入数据。除此之外，CPU还有一组寄存器，可以临时记住一些数据（就像大脑的临时记忆），但是存储量很有限。CPU的运算都只能用寄存器里的数据，要想接触内存，必须先从内存读入，计算完以后再写入到内存中去。

卡西欧的CLASSWIZ计算器（包括一代和二代）都使用了**nX-U16/100**微控制器（下称**nX-U16**）。因此，一代和二代的汇编语法是一样的。

* + 1. **通用寄存器**

由上小节所述，nX-U16拥有一些寄存器。它们可以分类为**通用寄存器**和**控制寄存器**。通用寄存器，就是说没有特殊的功能，可以随意使用。而控制寄存器是一类具有特殊的功能的寄存器，这些等后一小节讲常用指令时再一并讲。

nX-U16有16字节的通用寄存器。这些寄存器分别编号为R0、R1、……、R15，每个大小为1字节（等同于两个16进制数）。相邻的两个通用寄存器组合在一起，就拼出了一组2字节长的寄存器：ER0、ER2、……、ER14。其中ER12又叫BP(BasePointer)，ER14又叫FP(FramePointer)。也可以四个四个一拼，组成XR0、XR4、XR8、XR12。还能八个八个一拼，组成QR0和QR8。这些组合可以用一张图来表示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **QR** | **XR** | **ER** | **R** | **备注（别名）** |
| QR0 | XR0 | ER0 | R0 | 参数/返回值（见调用约定） |
| R1 | 参数/返回值（见调用约定） |
| ER2 | R2 | 参数/返回值（见调用约定） |
| R3 | 参数/返回值（见调用约定） |
| XR4 | ER4 | R4 |  |
| R5 |  |
| ER6 | R6 |  |
| R7 |  |
| QR8 | XR8 | ER8 | R8 |  |
| R9 |  |
| ER10 | R10 |  |
| R11 |  |
| XR12 | ER12 | R12 | BasePointer （低字节） |
| R13 | BasePointer （高字节） |
| ER14 | R14 | FramePointer （低字节） |
| R15 | FramePointer （高字节） |

因此，如果要更改R7的值，你可以直接更改R7，也可以更改它所属的寄存器组合，如ER6，XR4，甚至是QR0。这一点对于我们日后编写代码尤为有用。

表上未列出的寄存器是不能使用的。

寄存器组合的时候，数据的高位存在编号大的寄存器里，低位存在编号小的寄存器里。比如，如果ER0=0xDEAD，那么R0=0xAD，R1=0xDE。这种存储方式被称为**小端字节序**（Little Endian）。

* 1. **常用指令（nX-U16指令集）**

本节中我们将学习一些常用的指令。学会这些指令，我们就能看懂大多数代码了。（是不是很开心😊）

* + 1. **MOV和ADD**

MOV a, b表示：“将b赋值给a”，其中a必须是寄存器，b可以是一个数，也可以是寄存器。比如，MOV R0, 0x13表示把0x13赋给寄存器R0；MOV ER2, ER0表示把ER0的数值赋给ER2。

ADD a, b表示：“将a+b赋值给a”。它的语法和MOV相同。如果结果超过0xFF，那么就减去0xFF。

**小练习**：下面的程序运行完后，R0和R1分别等于几？

**注：**0x为16进制前缀。

MOV R0, 0x0

MOV R1, 0x1

ADD R0, R1

ADD R1, 0x1

ADD R0, R1

ADD R1, 0x1

答案：R0=0x3，R1=0x3。

* + 1. **CMP与寄存器PSW**

PSW是一个控制寄存器，称为程序状态字。正如它的名称，它存储着当前程序运行的状态。因此，在每一次CPU的运算之后，这些分区的值就会根据运算结果而更新。

PSW长一字节（8bit）。这个8位寄存器包含五个标志位（每个1bit），一个控制位（1bit）和一个字段（2bit）。

下图显示了PSW中的结构：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| PSW | C | Z | S | OV | MIE | HC | ELEVEL | |

* C表示加法中的进位（Carry），同时也表示减法中的借位。如果运算产生进位或借位，C=1。
* Z表示零（Zero）。如果结果是0x00，Z=1。
* S表示符号（Sign），它被赋值为运算结果的符号位。你可以简单地理解为：运算结果的最高位（如00101011的最高位是0，那么S=0）。

**选读：**一个二进制的数值，既可以表示非负整数，也可以表示有符号整数。有符号整数的最高位表示符号位，如果符号位是0，那数值就和非负的一样，如果符号位是1，就把这个数减去256。例如，二进制数01100001，不管解释成非负整数还是有符号整数，都是97。而11100001解释成非负整数是225，解释成有符号整数就是225-256=-31。

* OV表示有符号进位（Overflow）。它的运算规则较复杂。
* MIE表示CPU是否接受中断（外部输入）。
* HC表示半进位（Half-Carry），一般用不到。
* ELEVEL表示中断异常等级，CPU在接收到中断（外部输入）的时候设置为对应等级。

**简而言之，我们现在只需要记住前三个标志位C、Z、S的名称和作用。**

MOV a, b执行后，运算结果为a，PWS更新Z和S。

ADD a, b执行后，运算结果为a，PWS更新C、Z、S、OV和HC。

CMP a, b（Compare）表示：“比较a和b，结果存入PSW寄存器”。比较的结果为大于，等于或小于。实际上，CMP内部做了一次减法。因此a等于b就是Z=1，a小于b就是C=1。除此以外，CMP也会影响到PWS中的S。

**小练习：**下面每步操作后，PSW中C、Z、S的值分别为多少？（设最初C=0）

MOV R0, 0x7C

MOV R1, 0xD2

ADD R0, R1

CMP R1, R0

CMP R0, 0x4E

**答案：**

|  |  |  |
| --- | --- | --- |
| C | Z | S |
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 1 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |

* + 1. **B，BC与寄存器CSR:PC**

PC即程序计数器，长2字节（16bit）。与CSR一起构成CSR:PC(共2.5字节=20bit)CSR:PC指向CPU要执行的下一条指令。

CPU工作时，会从CSR:PC指向的地方取一条指令（**Fetch**），然后根据指令的长度，自动把PC+2或+4，让它指向下一条指令，接下来再执行最开始取来的那条指令。（一般来说，一条指令长2字节=16bit）

需要注意的是，PC自增不会影响CSR的值。

当CPU工作时，PC在没有分支指令的情况下会一直增长，而CPU就会按照顺序一条一条执行下去。

PC到达尾部会溢出重置(0xFFFE->0x0000)，如果有一条指令可以修改PC的值，CPU就会跳转到别的地方去。（分支指令）

如果不去修改PC的值，CPU就会按照顺序一条一条执行下去。如果有一条指令可以修改PC的值，CPU就会跳转到别的地方去。B xxx表示把“PC设为xxx”。比如，B 0x093FE表示跳转到093FE这个地方。

除此以外，还有一种条件跳转。BC cond, xxx（也写作Bcond xxx，两者都对）表示“如果满足条件cond，就跳转到xxx”。这条指令常常与CMP连用。

其中，条件cond有这些：

* AL表示无条件。从功能上看，BC AL, xxx和B xxx是相同的。
* EQ表示等于（也就是Z=1）。
* NE表示不等于（也就是Z=0）。
* LT表示小于（也就是C=1）。
* GE表示大于等于（也就是C=0）。
* LE表示小于等于（也就是C=1或Z=1）。
* GT表示大于（也就是C=0且Z=0）。
* cond还有GES、GTS、LES、LTS、PS、NS、NV、OV。它们用得较少，具体规则可以查阅操作手册，这里就不展开了。

接下来看一个例子。下面的程序意思是，如果R0在[0x20, 0x40)中，就赋值R1=1，否则，R1=0。

10000  CMP R0, 0x20

10002  BC LT, 1000C

10004  CMP R0, 0x40

10006  BC GE, 1000C

10008  MOV R1, 1

1000A  BC AL, 1000E

1000C  MOV R1, 0

**注：**指令前的数字（如1000A）表示指令的序号。

再看一个例子。下面的程序运行完后，R0等于1加到10的和55。（下面的数均为十进制，无0x前缀）

10000  MOV R0, 0

10002  MOV R1, 1

10004  ADD R0, R1

10006  ADD R1, 1

10008  CMP R1, 10

1000A  BC LE, 10004

* + 1. **L和ST**

L和ST负责对内存的读写。(Load/Store)

L a, p表示从内存中位置为p的地方读取数据并赋给a。其中a是寄存器，p可以是寄存器，也可以是数值。

比如，L R0, 0xD155表示从内存D155这里读取一个字节赋给R0，L ER2, [ER0]表示从ER0指向的地方读取两个字节赋给ER2。

如果需要读取或写入其他段（默认是第0段，段的概念见**2.2**），需要使用DSR前缀，用汇编是这么表示的：L ERn, DSR:[ERm]

ST的语法和L相同，功能正好相反，(从a读取数据并赋给内存中位置为p的地方)ST表示向内存写入。接下来看一个来自计算器ROM的实际例子。

0D074  MOV     ER8, ER0

0D076  MOV     ER10, ER2

0D078  MOV     ER12, ER0

0D07A  L       R0, [ER10]

0D07C  ST      R0, [ER12]

0D07E  ADD     ER10, 0x1

0D080  ADD     ER12, 0x1

0D082  MOV     R0, R0

0D084  BC      NE, 0D07A

**看完后，请同学们先仔细想一想它的功能。**

同学们发现了吗？它的功能是：从ER2指向的内存复制一些数据到ER0指向的地方，如果读到00，在复制完00后，就结束复制。（后面会说到：字符串是以NUL结尾的，因此这里其实就是复制了一个字符串。）

注意：0D082处的指令MOV R0, R0看起来没有用，但是，刚才说过，MOV指令也会更新Z，因此这里如果R0是00，Z就设为1，否则，Z设为0。下一条指令BC NE, 0D07A在这里意思就是“如果R0不是00，就跳转到0D07A”。

### BL，RT与寄存器LR

函数A要调用函数B，然后函数B返回到函数A继续执行，这是如何实现的？

第一步，肯定是直接跳转到那个函数的开始的地址，也就是B指令。但是，如果直接这么操作，会有问题——程序无法返回了。

为了解决这个问题，nX-U16提供了两条指令。BL xxx表示“设置LCSR:LR下一条指令的地址（也就是函数调用完，应该返回的地址，在上述情境中，为A的下一条指令），并跳转到xxx”。LR（链接寄存器，Link Register）和PC一样，是2字节长的控制寄存器。与LCSR构成LCSR:LR(共2.5字节=20bit)。它的功能就是备份CSR:PC。

当函数要返回时，RT表示“把LR赋给PC”，也就是跳转回LR指向的那条指令。

假设刚才那段程序后面是“0D086  RT”，那它就成了一个可以返回的函数。看这段代码：

MOV R0, 0x80

MOV R1, 0xD1 ; nX-U16不支持MOV ER0, 0xD180这样的指令，需要分开来写

MOV R2, 0x48

MOV R3, 0xD2

BL 0D076

这段代码调用了函数0D076。根据刚才提到0D076的功能，我们发现，这段代码可以从0xD248这里复制一个字符串到0xD180。

结合第四章的知识，0xD180是计算器的输入区，而0xD248是回放区。调用回放的时候就会做这么一次复制。所以我们已经帮卡西欧实现了一个功能！

小练习：回忆一下我们已学的函数及其功能。

1. MOV :
2. ADD :
3. CMP :
4. B :
5. BC :
6. L :
7. ST :
8. BL :
9. RT :

### 栈

#### PUSH，POP与寄存器SP

如果某个函数需要保存一些数据，当然可以通过寄存器，但是，寄存器是所有函数共享的（毕竟都在同一个CPU上运行），调用一个函数肯定会“破坏”特定的寄存器。

聪明的计算机科学家想到一个办法。他们创造了一种数据结构叫做“**栈**”（**Stack**），用来保存寄存器和其他临时数据。栈是内存中的一块特定区域。

栈直观来说可以理解成一叠纸。每调用一个函数，这个函数就在栈上加一张纸，它的全部数据都保存在这张纸上，当它退出时，就把这张纸拿掉，这样调用它的函数就能不受干扰地继续工作了。只要每个函数都只操作自己的这张纸，而且退出时拿掉自己的纸，数据就不会乱套。而且，可以发现程序运行时，永远在操作最上面的这张纸，编程的时候很方便。

一叠纸很明显有个“最上面这张”。因此CPU里有个寄存器SP（栈指针**Stack Pointer**,长度2字节），用来保存栈顶的位置（指示“最上面的纸”）。

另外，L和ST在栈上不太方便（因为存数的时候还要同时改变SP的位置），因此CPU有两条新的指令，PUSH和POP（推入和弹出）。

PUSH r表示“先把SP减去r的长度，然后把r存到SP指向的地方”，其中r是寄存器。POP r正好相反，表示“先把SP指向的数据赋给r，再把SP加上r的长度（栈的内存不变）”。

由于nX-U16处理的栈是“向下生长”的(“下”指内存编码小)，所以推入任何数据都会使SP数值减小。弹出任何数据都会使SP数值增大。

注意：nX-U16的栈是**两字节对齐**的，如果出入栈的数据不到两字节，就会补成两字节，如POP R0（1字节）实际上SP+2，PUSH LR实际上SP-4。

看这段代码：

MOV     R0, 0x80

MOV     R1, 0xD1

MOV     R2, 0x48

MOV     R3, 0xD2

PUSH    ER0

PUSH    ER2

POP     ER0

POP     ER2

运行的时候，一开始栈的下方是未初始化的（可以是任何数据），为了方便理解，我们假设SP以下的内存全是0。

.. 00 00 00 00 00 00[xx]xx ..

其中方括号表示SP指向的地方，xx表示上一个函数的栈。

在PUSH ER0时，由于ER0是两字节长，因此SP先会减去2，栈变为：

.. 00 00 00 00[00]00 xx xx .. //中间步骤

然后把ER0的值存到栈上，就是：

.. 00 00 00 00[80]D1 xx xx ..

同理，PUSH ER2后，栈变为：

.. 00 00[48]D2 80 D1 xx xx ..

接下来，POP ER0把栈上的数值0xD248赋给ER0，然后SP+2，变为：

.. 00 00 48 D2[80]D1 xx xx ..

同理，POP ER2后，ER2变为0xD180，栈变为：

.. 00 00 48 D2 80 D1[xx]xx ..

栈的情况恰好和原来相同。可是ER0和ER2的值交换了。这说明，在栈上保存数据时，先进栈的数据要后出栈，这样才能保持数据顺序不变。

#### 调用约定

在函数调用函数时，一个函数的返回值可能是另一个函数的自变量，这就要求了参数的传递。调用约定就是一个函数如何传参（传递参数），以及如何返回的标准。

**CCU8**（卡西欧用的LAPIS编译器）的调用约定规定：R0、R1、R2、R3和栈空间可以用来传参，并使用R0、R1、R2、R3来存储返回值，而剩余的寄存器要求保持不变。

为了达成以上要求，对于使用栈来存储本地变量的函数，CCU8会自动生成保存寄存器和栈指针的代码：

进入时:

PUSH       LR

PUSH       QR8

PUSH       XR4

MOV        ER14,SP

返回时：

MOV        SP,ER14

POP        XR4

POP        QR8

POP        PC

这样可以保证调用函数时寄存器和栈保持不变。

#### 多级函数调用

根据**1.2.5**节的知识，函数返回看的是LR里存的返回地址。这个方法对一层函数调用是有效的。可是如果存在多级函数调用怎么办？每次调用都会覆盖LR寄存器，这样的话只能保证最后一级的函数调用能够正常返回。

例如：P1调用P2,此时LR=P1调用的返回地址，然后P2调用P3，此时LR=P2调用的返回地址，当P3返回时，LR的值被赋值给PC寄存器，CPU从P2的返回地址开始执行，当P2返回到P1时，LR的值还是P2调用的返回地址（P1调用的返回地址在P2发生调用时被覆盖掉了），因此，P2函数永远无法返回。

这时就需要备份LR。在函数的开头加一句PUSH LR，结尾处把RT换成POP PC，只要确保所有的PUSH和POP全都配对，就可以任意调用函数了。

### 寄存器EA

对于大量数据的存取，CPU提供了一个EA寄存器，长2字节。和通用寄存器不同，L和ST对EA还有一个特殊的语法：

L r, [EA]和一般的L指令没什么区别，而L r, [EA+]表示，在读取完以后，还要让EA加上读取的长度，也就是自动指向下一个数据。同理，对于ST也有一样的语法。

另外，EA可以用LEA指令直接赋值。例如LEA [0xF800]就表示把EA设为0xF800。

# 硬件与系统架构

## nX-U16处理器

## 内存

## 自检与计算器版本号

同时按下q7|键（无论是否开机），你的991CN X都应显示为如图所示：

## ROM概述

## 矩阵键盘

矩阵键盘是一种电子线路，通常用于输入数据或控制电子设备。它由多个行和列的按键组成，形成一个二维排列的键盘结构。每个按键都位于特定的行和列交汇处，这种布局使得使用少量的引脚（通常为行和列的总数之和）就可以控制大量的按键。991CN X中应用了一个8行7列的矩阵键盘。

### 结构

矩阵键盘由多行和多列的按键组成。每个按键都与一个行线（称为KI）和一个列线（称为KO）相连（O，I分别为Output和Input的缩写）。

注：即使991CN X键盘排布外观不是矩形，其电路逻辑也是矩形。

以下是991CN X的键盘码对应表：（可以注意到有些节点未设按键）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **KO KI** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **8** | q | Q | E | $ | t |  |  |
| **7** | T | r | ! | R | y | [ |  |
| **6** | a | s | d | ^ | i | h |  |
| **5** | z | x | u | j | k | l | 0 |
| **4** | J | b | ( | ) | n | m | . |
| **3** | 7 | 8 | 9 | o | C |  | K |
| **2** | 4 | 5 | 6 | O | P |  | M |
| **1** | 1 | 2 | 3 | + | p |  | = |

例如：d键对应的行线，列线即为KI6，KO3。

### 工作原理

1. **扫描行和列**
2. 控制器会逐行逐列地扫描整个矩阵。当某个按键被按下时，它所在的行和列之间形成电路连接，这会使得该行，列电平相等。
3. 通常，控制器会通过以下步骤检测按键状态：

* 激活第一列（KO1），将这一列设置为低电平，其余列设置为高电平；
* 按KI值由大到小的顺序检查所有行，测试哪一行有信号返回（即低电平），这表示在当前行与该列交叉的按键被按下；
* 重复这个过程，依次激活每一列，直到所有行都被扫描完毕。再从第一列开始。

1. **识别按键**
2. 通过识别当前激活的行和有信号的列，控制器就可以确定是哪一个按键被按下了。例如，如果当前激活的是KO5，且在KI3检测到了低电平信号，那么按下的按键就是KI3,KO5相交处的键为C键。

处理器以非常高的频率来扫描矩阵键盘，以保证我们按下按键时，总是能够及时地检测到。由于处理器的运算速度非常快，因此可以轻易地让我们认为是“实时”的。

### 优先级

在991CN X上，若**同时**[[1]](#footnote-1)按下1键和2键，则屏幕上应显示1；**同时**按下1键和4键，则屏幕上应显示4。同学们不妨自己动手试一试。这是为什么呢？

先说结论：当一个KI同时被多个KO接通时，只有KO最小的有效；当多个KI同时被一个KO接通时，只有KI最大的有效，和KO相反。

**简记：先看左右，再看上下，左优于右，上优于下。**

举个例子：当同时有KO1→KI1（即按下1）和KO2→KI1（即按下2）时，只有前者有效，因为前者KO小。当同时有KO1→KI1（即按下1）和KO1→KI2（即按下4)时，只有后者有效，因为后者KI大。一定要注意KO和KI的优先级是相反的。

原因：为了保护电路，在两个完整的扫描周期之间，有一段相对较长（仍然很短[[2]](#footnote-2)）的空闲时间。因此，我们可以认为，当我们按下按键时，处理器总是处在间歇时间当中。开始扫描后，根据2.5.2节中的工作原理，键位的扫描顺序为：【(x,y)指(KIx,KOy)】

(8,1) (7,1) (6,1) (5,1) (4,1) (3,1) (2,1) (1,1)

(8,2) (7,2) (6,2) (5,2) (4,2) (3,2) (2,2) (1,2)

(8,3) (7,3) (6,3) (5,3) (4,3) (3,3) (2,3) (1,3)

(8,4) (7,4) (6,4) (5,4) (4,4) (3,4) (2,4) (1,4)

(8,5) (7,5) (6,5) (5,5) (4,5) (3,5) (2,5) (1,5)

(8,6) (7,6) (6,6) (5,6) (4,6) (3,6) (2,6) (1,6)

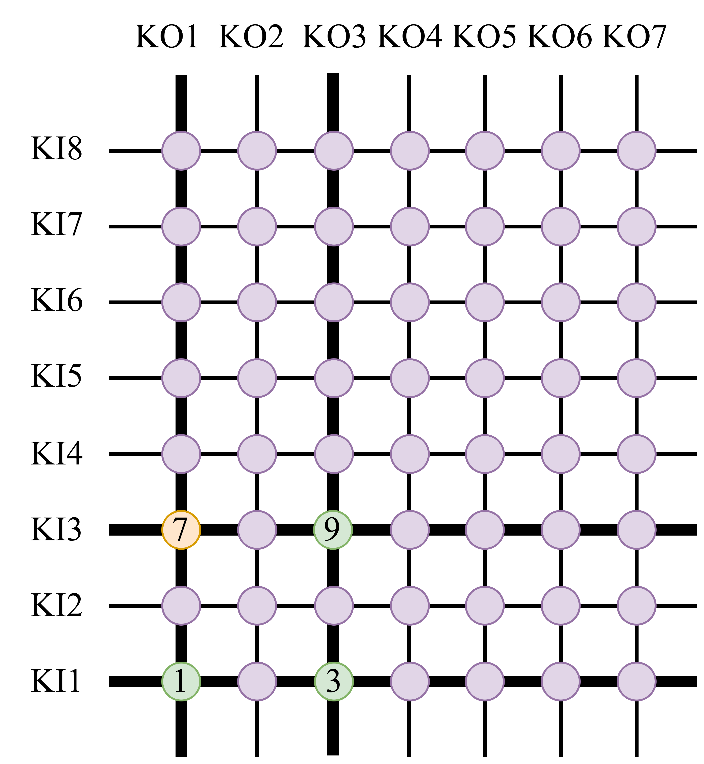
(8,7) (7,7) (6,7) (5,7) (4,7) (3,7) (2,7) (1,7)

对应键位图，该扫描顺序可简单概括为：**逐列由上到下扫描。**

当扫描到有键被按下后，处理器会直接开始新一轮扫描而非继续。故在上表中，越在前的节点对应的按键优先级越高，比如(KI5,KO2)在(KI3,KO7)的前面。故当同时按下x键和.键时，x的优先级更高。

### 鬼键问题（四按一松）

在上述的矩阵键盘设计中，如果有多个按键同时被按下，我们有时可以正确检测，有时却会发生错误，将未按下的按键误判为按下。这种现象被称为“鬼键”（Ghost Key）问题。

例如：**同时**按下1，3，9键时（“同时”的概念参考上页注释），则应显示7。

如图所示：当同时按下1，3，9键时，1键连通了KI1和KO1，3键连通了KI1和KO3，9键连通了KI3和KO3，这最终使得KI3和KO1相连通，造成了7键被按下的假象。由于7键的优先级>1，3，9键，故最终使得计算器认为按下了7键。

**想一想：同时**按下1，0，=，p，再松开1，屏幕上会出现什么？（“同时”的概念仍参考之前的注释）

### 自检

在2.3节中，我们讲过同时按下q7|键可以进入自检。事实上，进入自检的方式并不止这一种。

进入自检的条件是：**按下**|**（或刚装电池等原因导致开机）时同时存在KO1→KI3和KO1→KI8，且KO1的信号不再传入其它的KI。**

**想一想：**为什么同时按t，C，7，|也可以进入自检？

# 字符与表达式

## 单字节与双字节字符

字符的概念相信大家都已明白。字符在内存中储存，对卡西欧的程序员来说，最自然的想法就是每个字符占1字节，按序储存。但是，一个字节的数据只有162 = 256种可能。上代fx-ES型号里，就采取了每个字符1字节的办法，结果把控制字符、数字、变量、函数、算符，还有科学常数、单位换算全都挤在256个格子里，塞得满满当当。而现在的型号（CLASSWIZ X），有更多的科学常数和单位换算。（日版有200多个单位换算！）

那么单个字节是无论如何也塞不下了。是不是要全部改为双字节呢？那也不至于，因为双字节有65536种可能，用不了这么多，而且它会多占用一倍的空间。于是卡西欧的程序员采取了一种折中的办法，即同时采取单字节和双字节的字符。为了区分，规定双字节字符的第一个字节以F开头。以下是一些例子：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **字符** | 1 | Pa▸atm | rₑ | M+ |
| **码位** | 31 | FE1A | FD3A | FB1A |

**注：**如果要查看完整的字符表，请参见**附录一 字符表**。

特殊地，还要有个特殊字符来标记字符串的结束。它就是字符00。不过，为了方便，程序中许多操作都以字节为单位来进行，此时对程序来说，标记字符串末尾的就是字节00，也就是NUL，不同于字符00。这就导致像FE00这样的字符在程序的不同部分含义不一样，它有时只是一个一般的字符，有时却能标记字符串的结束。

一般来说这不要紧，因为这样的字符在正常情况下不会出现；反过来说，异常就出自这种特殊情况。但在探讨异常前，我们要先学习正常情况下各种操作的逻辑。

## 数学显示字符

|  |  |  |
| --- | --- | --- |
| 名称 | 自然书写模板 | 源码 |
| 带分数 |  | 18 1F 1D 1A a 1B 1A b 1B 1A c 1B 1E |
| 循环小数 |  | 2F 1A a 1B |
| 求和 |  | 50 1A a 1C b 1C c 1B |
| 积分 |  | 51 1A a 1C b 1C c 1B |
| 导数 |  | 52 1A a 1C b 1B |
| 连乘 |  | 53 1A a 1C b 1C c 1B |
| 绝对值 |  | 68 1A a 1B |
| e的幂 |  | 72 1A a 1B |
| 10的幂 |  | 73 1A a 1B |
| 平方根 |  | 74 1A a 1B |
| 对数 |  | 7D 1A a 1C y 1B |
| 分数 |  | C8 1D 1A a 1B 1A b 1B 1E |
| 幂 |  | a C9 1A b 1B |
| n次方根 |  | CA 1D 1A a 1B 1A b 1B 1E |

## 映射字符

# 内存分区

在991CN X中，计算器的内存可以被看作被分成了若干个分区，每一块区域都存储着与功能对应的数据。在这一章中，我们将学习每部分内存分区的功能。

## 输入区

输入区对应的内存从D180处开始，到D246处结束，共199个字符。通常来说，输入区中存储着当前输入的内容以及之前输入内容的“残留”。

下面，我们先来看一个例子，想一想输入区运作的规律。

1. 打出12345，此时D180处的内存为：

D180： 31 32 33 34 35

1. 删去

## 回放区

回放区对应的内存从D248处开始，到D30E处结束，共199个字符。

## 撤销区

撤销区对应的内存从D522处开始，到D5E8处结束，共199个字符。

## 变量区与变量存储格式

## 历史记录区

## 设置区

## 即时状态区

## 堆栈区

## 随机数种子

## 显示缓存区

## 空闲区

## 杂项

# 输入的显示与编辑

## 输入的显示

## 输入的编辑

### 光标移动

### 增删字符

### 撤销

### 插入

### 编辑数学表达式

# 特殊字符

## 字符转换器

### 字符转换器原理

### 连续刷字符

### 根号法复制字符

## 溢出

### 达成溢出的方法

#### 方程溢出

#### 线性求和溢出

#### 双字节溢出

#### 框溢出

### 溢出法刷字符

### 溢出法复制字符

### 定向填充法

## @法刷字符

### @法刷字符原理

### @法的优化

### 在x an模式下@法的变化

## 字符的特殊性质

# 返回导向编程

## an/@的异常性质

## ROP进入方法

### x an/@模式

### an+12char法与@+2char法

## 简单ROP

### 内存修改器

### 显示字符

### 读取按键

### 循环结构

### 分支结构

## ROP进阶

### 编辑内存

观察ROM中0D070

### 计时

### ROM提取

### 绘制屏幕

### 调用其他函数

### quickcpy——快速注入内存

## ROP实战

### 内存注入可视化

### 像素编辑器

### 贪吃蛇

### 俄罗斯方块

### 1024

### 函数绘图

# ROM刷机

# 附录一 字符表

此处有fx-991CN X 硬件版本Ver C和Ver F的字符表。

# 附录二 常用汇编指令

### RB、SB、TB

这三条指令用来按bit操作数值。

TB a.x表示检测a的第x个二进制位，并按照结果设置Z。其中，最低位是0，最高位是7。a可以是寄存器，也可以是内存地址。

RB和SB语法相同，RB表示把这一位设为0，SB表示把这一位设为1。

##### 例：

MOV     R0, 11010010b

RB      R0, 1

SB      R0, 3

执行完后，R0=11011001b

# 附录三 按键，键盘码，内码表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **KI KO** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **8** | q | Q | E | $ | t |  |  |
| **7** | T | r | ! | R | y | [ |  |
| **6** | a | s | d | ^ | i | h |  |
| **5** | z | x | u | j | k | l | 0 |
| **4** | J | b | ( | ) | n | m | . |
| **3** | 7 | 8 | 9 | o | C |  | K |
| **2** | 4 | 5 | 6 | O | P |  | M |
| **1** | 1 | 2 | 3 | + | p |  | = |

# 附录四 模拟器的配置与使用

为了更好地研究计算器，我们需要有模拟器。以下是由编写组人员@qiufuyu123制作的991CN X模拟器，源码位于Github上。

Github地址：[github.com/qiufuyu123/CasioEmuNeo/releases/latest](https://github.com/qiufuyu123/CasioEmuNeo/releases/)

# 附录五 ROM函数地址索引

# 附录六 用Ghidra反编译ROM

# 附录七 Github上的数字资源仓库

本书编写组在Github上存放了一个数字资源仓库，里面存储了所有本书所需的资源。

网址： [github.com/Physics365/991CN-X-CW-Decompilation](https://github.com/Physics365/991CN-X-CW-Decompilation)

# 结 语

1. 一般来说，同时不易做到，此时可以在关机状态下按住1键和2键（不分先后），再开机，则应显示1。 [↑](#footnote-ref-1)
2. 一个完整的扫描周期大约为1μs的数量级，而间歇时间约为1ms的数量级。 [↑](#footnote-ref-2)